MATH491

Summer Research Project

2006– 2007

Pollard's rho-Algorithm, And Its Applications to Elliptic Curve Cryptography

Stephen McConnachie

Department of Mathematics and Statistics
University of Canterbury

MATH 491: Summer Research Project

"Pollard's rho-Algorithm, And Its Applications To Elliptic Curve Cryptography"

Stephen McConnachie 52006338

Abstract

This project aims to describe Pollard's rho-Algorithm for solving the Discrete Logarithm Problem in a group, and to look at how this algorithm applies to Elliptic Curve Cryptography. We start with a general definition of Elliptic Curves and the Discrete Logarithm Problem, and go on to describe Pollard's rho-algorithm in detail. We prove some of the assumptions used in the algorithm description, and in the final section we look at some of the issues with which one is faced when trying to apply the algorithm to the Elliptic Curve Discrete Logarithm Problem.

Introduction

Cryptography is the study of making and breaking codes. A *cryptosystem* is a procedure for keeping information secure (for a formal definition of a cryptosystem, see [1] p71, Definition 3.1.1). In particular, this paper deals with *public-key cryptosystems* which are cryptosytems where only one person, say Alice, knows how to decrypt ciphertext (encrypted messages), but anyone can encrypt a plaintext (unencrypted message) using a *public key* published by Alice. The public key reveals no information about the private key that Alice uses to decrypt the ciphertext.

This paper explores an algorithm for breaking a common type of cryptosystem. The first section looks at an example of a cryptosystem, and describes a method known as Elliptic Curve Cryptography. The second section looks at an algorithm called Pollard's rho-algorithm, which can be used to break the cryptosystem; section three contains some proofs and remarks, while the final section looks at how we can apply this code-breaking algorithm to Elliptic Curve Cryptography.

Section One: Elliptic Curves and the Discrete Logarithm Problem

1.1 The Discrete Logarithm Problem

Many public-key cryptosystems are based on the difficulty of what is known as the *Discrete Logarithm Problem* (DLP). Suppose we have a prime p and a generator γ for the group $\mathbb{Z}_p^*$, that is, an element $\gamma \in \mathbb{Z}_p^*$ such that the set $\{\gamma^r \bmod p \mid 1 \leq r \leq p-1\}$ contains every non-zero element of $\mathbb{Z}_p^*$. The DLP is calculating the index x in the equation

$$\gamma^x \equiv \alpha \bmod p \qquad \text{(Eqn 1.1)}$$

for known γ, p and α. A deterministic brute-force approach will always solve the problem, but for large p it is computationally impractical to calculate $\gamma^r \bmod p$ for $1 \leq r \leq p-1$ and compare the result to the known α.

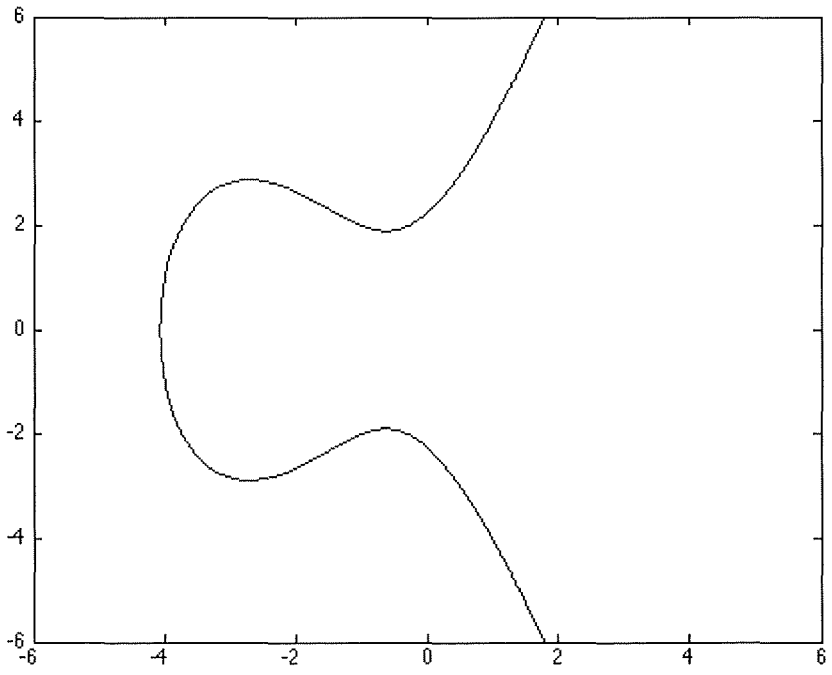
Note that we refer to the DLP as calculating *the* index; there is only one possible element $x \in \mathbb{Z}_p^*$ that satisfies Eqn 1.1. The generator γ, by definition, generates every element of $\mathbb{Z}_p^*$, and γ is a generator if and only if $|\gamma| = |\mathbb{Z}_p^*|$; therefore every element in $\mathbb{Z}_p^*$ appears exactly once in the set $\{\gamma^r \bmod p \mid 1 \leq r \leq p-1\}$, and there is only one possibility for x.

1.2 Elliptic Curves

Given some field **F**, an Elliptic Curve $E(\mathbf{F})$ is the set of all solutions to an equation of the form

$$y^2 = ax^3+bx^2+cx+d \qquad \text{(Eqn 1.2)}$$

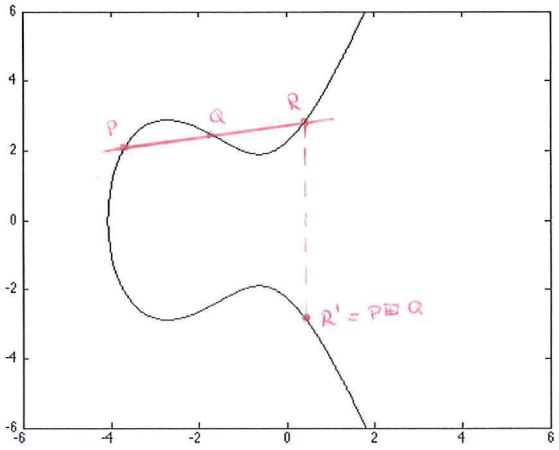
together with a point ∞ not on the curve, where $a,b,c,d \in \mathbf{F}$, $a \neq 0$, and the cubic in x has three distinct roots; that is, $E(\mathbf{F})$ is the set $\{(x,y) \in \mathbf{F} \times \mathbf{F} \mid y^2 = ax^3+bx^2+cx+d\} \cup \{\infty\}$. [3 pg4] When $\mathbf{F}=\mathbb{R}$, the curve has a graph that is symmetric about the x-axis as shown in Fig 1.3.



$y^2 = x^3 + 5x^2 + 5x + 5$

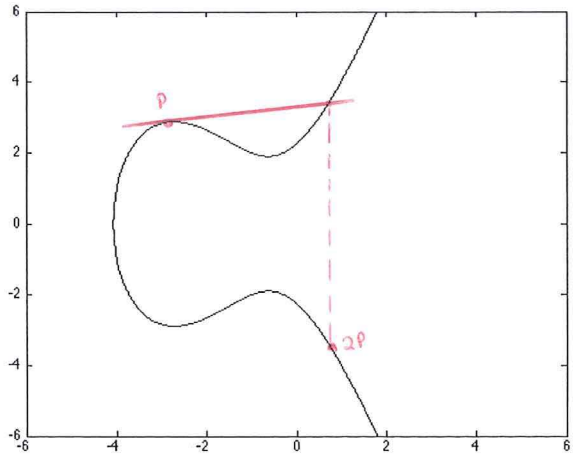
(Fig 1.3)

The points of an elliptic curve, together with a point ∞ not on the curve, form an abelian group under an operation known as the 'addition law', denoted by ⊞. [3 pg11, Theorem 3.2]. For an explanation (but no formal proof) of why this is true, see Garrett's text [2 Section 28.3, pg452]. This addition law can be described geometrically in the real number case from the graph. Let $P = (x,y)$ be a point on the curve; then we define $P' = (x,-y)$ to be its reflection in the x-axis. Taking two points P, Q on the curve, the third point $R' = P \boxplus Q$ is found as follows: drawing a straight line between P and Q, we take the third intersection of the line and the curve and call it R. Then $P \boxplus Q$ is the reflection R' of R in the x-axis. At this point we can note that the group $E(\mathbb{R})$ will be abelian by construction; joining two points with a line is not affected by the order in which the points are presented. The proof of commutativity in the general case $E(\mathbf{F})$ is beyond the scope of this research project.



Addition law in Elliptic Curves

(Fig 1.4)



Doubling a point in Elliptic Curves

(Fig 1.5)

There are some special cases: when $P = \infty$, $\infty \boxplus Q = Q$, and similarly when $Q = \infty$, $P \boxplus \infty = P$. When $P = Q$, we define $P \boxplus Q = 2P$ as shown in Fig 1.5; we take the tangent line at the point P and find the second intersection between the line and the curve. Given the point $P = (x,y)$, if $y=0$, $2P = \infty$.

1.3 Algebraic Description Of The Addition Law

This description gives us an idea of how the addition law works, but it does not work when the elliptic curve is over a finite field. In this case the graph is not continuous (the graph is in fact just a finite set of points, symmetric about the x-axis) and drawing lines to intersect the curve does not make sense. We must therefore describe the addition law using algebra, as follows. [3 pg10]

Let $P_1, P_2 \in E(\mathbb{Z}_p)$. Define

$$P_1 \boxplus P_2 = \begin{cases} P_1 & \text{if } P_2 = \infty \\ P_2 & \text{if } P_1 = \infty \end{cases}$$

Otherwise, write $P_1 = (x_1,y_1)$, $P_2 = (x_2,y_2)$. Then

$$P_1 \boxplus P_2 = \begin{cases} \infty & \text{if } x_1 = x_2 \text{ and } y_1 \neq y_2 \\ \infty & \text{if } x_1 = x_2 \text{ and } y_1 = y_2 = 0 \end{cases}$$

Otherwise, $P_1 \boxplus P_2 = P_3 = (x_3,y_3)$, where

$$x_3 = -x_1 - x_2 - ba^{-1} + m^2a^{-1},$$
$$y_3 = -y_1 + m(x_1 - x_3)$$

and

$$m = \begin{cases} (y_2 - y_1)(x_2 - x_1)^{-1} & \text{if } P_1 \neq P_2 \\ (3ax_1^2 + 2bx_1 + c)2^{-1}y_1^{-1} & \text{if } P_1 = P_2 \end{cases}$$

Here x_i, y_i, $m \in G$. If $G = \mathbb{Z}_p$, then every non-zero element of G has an inverse, because p is prime; we assume that $p \neq 2$ so that the term 2^{-1} in the last formula makes sense. [3 pg11]

We use additive notation, rather than the usual multiplicative notation, for the group $E(\mathbf{F})$; for example, we write $P \boxplus Q$ instead of PQ, we write $-P$ rather than P^{-1} for the inverse of P, and call $-P$ the *negative* or *additive inverse* of P. We also write nP instead of P^n for n an integer. [3 pg12]

1.4 The Discrete Logarithm Problem with Elliptic Curves

The group $G = E(\mathbf{F})$ is not cyclic in general; that is, there is not necessarily an element $P \in E(\mathbf{F})$ that generates every element of the group by repeated addition to itself. Let $P \in G$ be a point on the curve, and let $G' \subseteq G$ be the set $\{\infty, \pm P, \pm 2P, \pm 3P, \ldots\}$. The *Elliptic Curve Discrete Logarithm Problem* (ECDLP) is still considered difficult when $|G'|$ has large order.

The ECDLP describes the problem of finding x in the equation

$$Q = xP \qquad \text{(Eqn 1.6)}$$

where $Q \in G' \subseteq E(\mathbf{F})$, $P \in E(\mathbf{F})$ and $x \in \mathbb{N}$.

1.5 Elliptic Curve Cryptography

Many public-key cryptosystems based on group operations in $\mathbb{Z}_p^*$ can be modified to use the addition law in $E(\mathbf{F})$. For example, the El Gamal cryptosystem [8] is a public-key cryptosystem that relies on the difficulty of the DLP in a finite field (usually $\mathbb{Z}_p$ for large prime p). El Gamal can be directly applied to Elliptic Curves by selecting an appropriate cyclic subgroup of the set of points on the curve.

El Gamal cryptosystem comparison: Suppose Bob wants to send secure messages to Alice, so that only Alice can decrypt the message.

$\mathbb{Z}_p^*$

cyclic group G of order q, generator $\gamma \in G$

Alice's private key is $a \in \{0,1,2,...,q\text{-}1\}$

Alice computes $A := \gamma^a$

Her public key[1] is (G, q, γ, A)

Bob takes plaintext $m \in G$. He takes random $b \in \{0,1,...,q\text{-}1\}$

calculates $c_1 := \gamma^b$ and $c_2 := mA^b$

sends (c_1, c_2) to Alice

Alice performs the calculation

$$\begin{aligned} c_2 c_1^{-a} &= c_2 (c_1^a)^{-1} \\ &= mA^b (\gamma^b)^{a})^{-1} \\ &= m\gamma^{ab}\gamma^{-ab} \\ &= m \end{aligned}$$

For example:

Working in $\mathbb{Z}_{17}^*$ with generator 3, Alice chooses random $a=12$. She computes $A = 3^{12} \bmod 17 \equiv 4$. Her public key is now $(G, q, \gamma, A) = (\mathbb{Z}_{17}^*, 16, 3, 4)$

(although in the case of $G=\mathbb{Z}_p^*$, q is implied and does not need to be made public)

Bob takes plaintext $m=10$, and chooses random $b=11$. He calculates

$$c_1 = 3^{11} \bmod 17 \equiv 7$$

and

$$c_2 = 10.4^{11} \bmod 17 \equiv 10.13 \bmod 17 \equiv 11$$

and sends (c_1, c_2) to Alice.

Alice computes $c_2 c_1^{-a}$

$$\begin{aligned} &\equiv c_2 (c_1^a)^{-1} \bmod 17 \\ &\equiv 11.13^{-1} \bmod 17 \\ &\equiv 11.4 \bmod 17 \\ &\equiv 10 = m \end{aligned}$$ □

$E(G)$

abelian group $E(\mathbb{P})$ for some field $\mathbb{P}$, group not necessarily cyclic but has a cyclic subgroup G of "large" order q, with generator $\gamma \in G$

Alice's private key is $a \in \{0,1,2,...,q\text{-}1\}$

Alice computes $A := a\gamma = \gamma \boxplus \gamma \boxplus \gamma \boxplus ... \boxplus \gamma$, a times

Her public key is (E, q, γ, A)

Bob takes plaintext $m \in G$. He takes random $b \in \{0,1,...,q\text{-}1\}$

calculates $c_1 := b\gamma$ and $c_2 := m \boxplus bA$

sends (c_1, c_2) to Alice

Alice performs the calculation

$$\begin{aligned} c_2 \boxplus (-ac_1) &= (m \boxplus bA) \boxplus (-ab\gamma) \\ &= m \boxplus ba\gamma \boxplus (-ab\gamma) \\ &= m \end{aligned}$$

1 *"Her public key..."* Note that in both cases, a complete list of elements in G and E are not required; only a description of the group and the group operation are required for the public key. For example, "$G=\mathbb{Z}p^*$, with the standard modular arithmetic", "E is the curve $y^2 = x^3 + 4x^2 + 4x + 3$, with the standard elliptic curve addition law".

Section Two: Pollard's rho-Algorithm

Pollard's rho algorithm [7] is a method for solving the Discrete Logarithm Problem. It has a runtime of $O(\sqrt{|G|})$, which means the amount of processor time required to run the algorithm is less than or equal to some constant k multiplied by $\sqrt{|G|}$. This is the same runtime as that of Shanks' Baby-Step Giant-Step algorithm [1 pp214-217] for finding the index in the DLP, but the rho algorithm requires constant storage (i.e. O(1)). This makes Pollard's rho algorithm much more space-efficient than Shanks' algorithm, which requires $O(\sqrt{|G|})$ storage. We will cover the storage requirements for Pollard's algorithm in more depth in the next section.

2.1 Algorithm Description

Recall that the DLP is solving the equation $\gamma^x \equiv \alpha \mod p$ (Eqn 1.1) for x. We define Pollard's rho-algorithm in the following way (based heavily on Buchmann's description of the algorithm [1 pp217-221]). Let G be a finite cyclic group of known order n, and $\gamma \in G$ be a generator for this group. Define G_1, G_2, G_3 to be three pairwise disjoint subsets of G such that $G_1 \cup G_2 \cup G_3 = G$; for example divide G into three subsets of roughly equal size.

2.1.1 The β-Algorithm

Define the function f: $G \rightarrow G$ by the piecewise expression

$$f(\beta) := \begin{cases} \gamma\beta & if\ \beta \in G_1 \\ \beta^2 & if\ \beta \in G_2 \\ \alpha\beta & if\ \beta \in G_3 \end{cases}$$

We choose a random number x_0 in the set $\{1,...,n\}$ and compute the group element $\beta_0 = \gamma^{x_0}$. We then compute the recursive sequence $(\beta_i)_{i \geq 0}$

$$\beta_{i+1} = f(\beta_i).$$

The elements of this sequence can be written as

$$\beta_i = \gamma^{x_i}\alpha^{y_i}, \quad i \geq 0.$$

We define x_i and y_i recursively in a similar manner to our function f:

$$x_{i+1} = \begin{cases} x_i + 1 \mod n & \text{if } \beta_i \in G_1 \\ 2x_i \mod n & \text{if } \beta_i \in G_2 \\ x_i & \text{if } \beta_i \in G_3 \end{cases}$$

and

$$y_{i+1} = \begin{cases} y_i & \text{if } \beta_i \in G_1 \\ 2y_i \mod n & \text{if } \beta_i \in G_2 \\ y_i + 1 \mod n & \text{if } \beta_i \in G_3 \end{cases}$$

where x_0 is the initial random number and $y_0 = 0$.

2.1.2 Proof of the β-Algorithm Recursive Definitions

Buchmann does not explain why β can be expressed in terms of x and y, but the proof is fairly straightforward. From the definition of $f(\beta)$ and since $\beta_0 = \gamma^{x_0}$, then every β_i will be of the form $\gamma^{x_i}\alpha^{y_i}$. Then, to derive our recursive definitions of x_i and y_i, consider the three cases $\beta_i \in G_1$, $\beta_i \in G_2$, $\beta_i \in G_3$.

Case 1: $\beta_i \in G_1$

Then $\beta_{i+1} = f(\beta_i) = \gamma\beta_i$

since $\beta_i = \gamma^{x_i}\alpha^{y_i}$, $\beta_{i+1} = \gamma\beta_i = \gamma^{x_i+1}\alpha^{y_i}$

but $\beta_{i+1} = \gamma^{x_{(i+1)}}\alpha^{y_{(i+1)}}$

$\therefore$ $x_{i+1} = x_i + 1 \mod n$, $y_{i+1} = y_i \mod n$

Case 2: $\beta_i \in G_2$

Then $\beta_{i+1} = f(\beta_i) = \beta_i^2$

since $\beta_i = \gamma^{x_i}\alpha^{y_i}$, $\beta_{i+1} = \beta_i^2 = (\gamma^{x_i}\alpha^{y_i})^2 = \gamma^{2(x_i)}\alpha^{2(y_i)}$

but $\beta_{i+1} = \gamma^{x_{(i+1)}}\alpha^{y_{(i+1)}}$

$\therefore$ $x_{i+1} = 2x_i \mod n$, $y_{i+1} = 2y_i \mod n$

Case 3: $\beta_i \in G_3$

$$\text{Then} \quad \beta_{i+1} = f(\beta_i) = \alpha\beta_i$$

$$\text{since} \quad \beta_i = \gamma^{x\_i}\alpha^{y\_i}, \quad \beta_{i+1} = \alpha\beta_i = \gamma^{x\_i}\alpha^{y\_i+1}$$

$$\text{but} \quad \beta_{i+1} = \gamma^{x\_(i+1)}\alpha^{y\_(i+1)}$$

$$\therefore \quad x_{i+1} = x_i \bmod n, \qquad y_{i+1} = y_i + 1 \bmod n$$

■

2.1.3 Computing The Index From The β Match

Since we are working in a finite group, two elements in the sequence $(\beta_i)_{i\geq 0}$ must be equal; that is, there exists $i \geq 0$ and $k \geq 1$ with $\beta_{i+k} = \beta_i$. This implies

$$\gamma^{x\_i}\alpha^{y\_i} = \gamma^{x\_(i+k)}\alpha^{y\_(i+k)}$$

$$\Rightarrow \quad \gamma^{x\_i - x\_(i+k)} = \alpha^{y\_(i+k) - y\_i}$$

$$\Rightarrow \quad \gamma^{x\_i - x\_(i+k)} = (\gamma^{x})^{y\_(i+k) - y\_i} \qquad \text{(by Eqn 1.1)}$$

$$\Rightarrow \quad \gamma^{x\_i - x\_(i+k)} = \gamma^{x[y\_(i+k) - y\_i]}$$

$$\Rightarrow \quad x_i - x_{i+k} = x(y_{i+k} - y_i) \bmod n \qquad \text{(Eqn 2.1)}$$

(see [1] Corollary 2.9.3 for proof that this step works in an abelian group)

We now have a congruence equation that we can solve for x, and thus solve the Discrete Log Problem (Eqn 1.1). The solution is unique mod n if $y_i - y_{i+k}$ is invertible mod n. If the solution is not unique, we test the different possibilities for x to see which satisfies Eqn 1.1; if this is computationally impractical (if there are too many possibilities), then the algorithm is applied again with a different initial x_0.

Buchmann's description of the algorithm does not include how to find the possibilities for x; he only shows the process by example ([1] Example 10.4.1). Following is a general-case description of the process by which we find all the possibilities for x.

Calculate $g := \gcd(y_i - y_{i+k}, n)$. If g divides $x_i - x_{i+k}$, then Eqn 2.1 has a unique solution mod n/g. If g does *not* divide $x_i - x_{i+k}$, retry the algorithm with a different initial x_0. To find the index x, we solve the congruence

$$[(x_i - x_{i+k})/g].z \equiv (y_i - y_{i+k})/g \bmod n \qquad \text{(Eqn 2.2)}$$

for z. Then the possibilities for x are the g solutions to the equation

$$z + k.(n/g), \quad 0 \leq k \leq g \qquad \bmod n$$

2.2 Storage Requirements – The One-Triplet Method

Note that if every triplet (β_i, x_i, y_i) is stored, the algorithm requires $O(\sqrt{|G|})$ storage. It is possible however to store only one triplet, and still guarantee a match is found.

We store our initial triplet (β_0, x_0, y_0). Now suppose the triplet $\tau_i = (\beta_i, x_i, y_i)$ is stored. Then we compute, compare with τ_i, and discard every triplet between (β_{i+1}, x_{i+1}, y_{i+1}) and (β_{2i-1}, x_{2i-1}, y_{2i-1}), stopping if a match is found and storing the final matching triplets. If no match is found, we discard τ_i, compute and store τ_{2i}, and repeat the compute-and-compare process. Fig 2.3 is an example of the stored triplets (using the same γ, α, p and x_0 values as Buchmann's Example 10.4.1, but computed with the algorithm in Appendix).

```
>> pollardZ(5,3,2017,1023)
      j   B_j   x_j   y_j
      0   986  1023     0
      1     2    30     0
      2    10    31     0
      4   250    33     0
      8  1366   136     1
     16  1490   277     8
     32   613   447   155
     64  1476  1766  1000
     98  1476   966  1128
```

(Fig 2.3)

So we see that only triplets with $i=2^{\ell}$ for some $\ell \in \mathbb{Z}$ are stored, and a match is found at $i=98$ between $i=2^6$ and $i=2^7$. A proof that a match occurs and will be found using this method of storing one triplet at a time, will be in the next section.

Section Three: Why The rho-Algorithm Works

3.1 β Match Guaranteed

The Pollard rho-Algorithm will always find a match: since we are computing an infinite sequence of β_i values, all of which are elements of a finite group, there will definitely be a match. Moreover, we need only $O(|\sqrt{G}|)$ sequence elements to make the probability for a match greater than $\frac{1}{2}$ (This can be shown by using the so-called "birthday paradox" - see Buchmann [1 pp118-119] for a proof).

3.2 Proof of One-Triplet Method

This algorithm will always find a match even with the storage modification (Section 2.2). The proof of this involves two parts: firstly we show that the sequence $(\beta_i)_{i\geq 0}$ is periodic after a match occurs, and secondly we show that we can still guarantee that a match is found when we only store one triplet at a time.

Proof that the sequence is periodic:
Let $\beta_s = \beta_{s+k}$ be the first match, which is not necessarily found using the one-triplet method. Then $\beta_{s+1} = \beta_{s+k+1}$ since the recursive construction of β_{i+1} depends only on β_i, and similarly $\beta_{s+\ell} = \beta_{s+k+\ell}$ for some integer $\ell \geq 0$. Thus the sequence is periodic after β_s, and we see why the algorithm is called the 'rho' algorithm – the 'tail' of the rho is the sequence β_0, $\beta_1,\ldots,\beta_{s-1}$, and the 'loop' of the rho is the periodic sequence β_s, $\beta_{s+1},\ldots,\beta_{s+k-1}$. ■

Proof that the match will be found using Buchmann's one-triplet method:
If $i = 2^\ell < s$ then no match is found; we replace $i = 2^\ell$ with $i = 2^{\ell+1}$. If $i = 2^\ell \geq s$ then β_i is in the periodic part of the sequence $(\beta_i)_{i\geq 0}$. If $i = 2^\ell \geq k$ (recall that k is the length of the period) then the sequence

$$\beta_{i+1}\ ,\ \beta_{i+2}\ ,\ \ldots\ ,\ \beta_{2i}$$

is at least as long as the period, so one if its elements will match $\beta_i = \beta_{2i}$. Now recall that after we store the triplet containing $\beta_i = \beta_{2^\wedge\ell}$, we compute this same sequence and compare its elements with β_i; hence a match *will* be found using this method of storage. ■

3.3 Remarks on One-Triplet Method and β-Algorithm

At this stage it is worth noting that our one-triplet method will not necessarily find the first occurrence of a match in the sequence $(\beta_i)_{i\geq 0}$. Our assertion is that this does not affect the final result x, nor does it significantly affect the runtime of the algorithm. This is based on results rather than a formal proof; the algorithm returns the correct index x with the one-triplet method, and finishes reasonably quickly.

In fact, it is very interesting that while writing the code for pollardZ.m (Appendix), even though the β-algorithm (Section 2.1.1) initially did not work (due to incorrect code it returned an incorrect sequence of triplets after the initial triplet), the correct index was still returned. When using the values from Buchmann's Example 10.4.1, the correct value of z

in Eqn 2.2 was also returned.

The following data is taken from a diary file that recorded some early code test-runs:

```
pollardZ(5,3,2017,1023)

 j  B_j  x_j  y_j
 0  986 1023    0
 1   10   31    0
 2   50   32    0
 4 1250   34    0
 8   64  136    2
16  436  277    9
32 1048  448  155
64  394 1766 1001
98  394  966 1129

ans =

    1030
```

(Fig 3.1)

Comparing this to Fig 2.3 (the correct list of triplets), we see that every triplet after the initial triplet is wrong; and yet, the algorithm finds a match in the same number of iterations (namely, i=98), and the final answer is correct, namely x=1030.

The reason for the error in the code is that the counter for the iterative beta algorithm was at the end of the loop instead of the beginning; the result being that the second triplet was computed but not stored, then the third triplet was stored as the second triplet and the error compounded.

This warrants further research as to how the β-algorithm can be varied without affecting the results, with applications to making the β-algorithm more efficient. This will also be covered briefly in the next section.

Section Four: Pollard rho-Algorithm and Elliptic Curves

Now we look at how Pollard's rho algorithm can be applied to solving the Discrete Logarithm Problem in Elliptic Curves. Several issues arise when trying to do this. For example, how do we define the three subsets G_1, G_2 and G_3? If we have to store them explicitly (as lists of elements) we might as well solve the DLP by enumeration, and the storage requirement for the subsets will be $O(n)$. When using the rho algorithm in $\mathbb{Z}_p$, we only need to store the two 'dividing points', $\frac{1}{3}.p$ and $\frac{2}{3}.p$ rounded to the nearest integer. Can we choose the subsets arbitrarily? What effect will different choices for the subsets have on the efficiency of the algorithm? We know that in the general case some care must be taken when choosing the subsets, for example $1 \notin G_2$. Are there any more cases, specific to elliptic curves, that will prevent success or slow the algorithm?

Another issue is that the arithmetic with the exponents x_i and y_i is performed mod n, where n is the order of the group. In practice, how do we know n for the elliptic curve $E(\mathbf{F})$?

The following is taken and adapted from Menezes' paper [5] on Elliptic Curve Cryptography:

If we assume that a 1 MIPS (Million Instructions Per Second) machine can perform 4×10^4 elliptic curve additions per second; this is a conservative estimate. The number of elliptic curve additions (*steps*) that can be performed by a 1 MIPS machine in one year is roughly 2^{40}.

Let n be the order of a cyclic subgroup $G' \subseteq E(\mathbf{F})$. Fig 4.1 shows, for various values of n, the computing power required to compute a single discrete logarithm using the Pollard rho-method. A *MIPS year* is equivalent to the computational power of a computer that is rated at 1 MIPS and utilized for one year.

Field size (in bits)	Size of n (in bits)	Estimated number of steps	MIPS years
163	160	2^{80}	9.6×10^{11}
191	186	2^{93}	7.9×10^{15}
239	234	2^{117}	1.6×10^{23}
359	354	2^{177}	1.5×10^{41}
431	426	2^{213}	1.0×10^{52}

Computing power required to compute elliptic curve logarithms with the Pollard rho-method.

Fig. 4.1

Conclusion

There is much more to applying Pollard's rho-algorithm to Elliptic Curves than the time constraints on this project allow us to cover. Further research is required.

There are more efficient versions of the Pollard rho-algorithm. See Menezes' reference [4 pg106] to Floyd's cyclefinding algorithm, as an alternative to our one-triplet method. Teske [9] experimented with different iterating functions – alternatives to our function f in section 2.1.1 – and found a function which significantly improved the runtime of the rho-algorithm. In 1999 van Oorschot and Wiener [10] showed how Pollard's rho algorithm can be parallelized so that when the algorithm is run in parallel on n processors, the expected runtime of the algorithm is reduced by roughly n times; according to Menezes [6 pg9], in 2001 parallelized Pollard's rho-algorithm was the fastest known general-purpose method for solving the Elliptic Curve Discrete Logarithm Problem.

Appendix

This appendix contains MATLAB code for algorithms used in my research to compute examples in the report. The first four are small function algorithms that the fifth and final algorithm, pollardZ.m, invokes while finding the index *x* in the DLP (Section 1.1).

invmod.m *code written by Hamish Silverwood*

Algorithm for calculating the inverse of an element mod *m*

```
function [X] = invmod(a, m)
%Check relative primality
if gcd(a,m)~=1
    error('a and m must be coprime (gcd(f,m)=1) for a to be invertible in Zm')
end

%Initiate Variables
xO=a;  yO=m;  MO=[1 0 0 1];  i=0;  yN=1;

while yN~=0
    qN=floor(xO/yO);
    xN=yO;
    yN=xO-qN*yO;
    MN=[MO(3) MO(4) (MO(1)-qN*MO(3)) (MO(2)-qN*MO(4))];
    xO=xN;
    yO=yN;
    MO=MN;
end

%Check for negatives
if MN(1)<0
    X=m+MN(1);
else
    X=MN(1);
end

return
```

iseven.m *code written by Hamish Silverwood, modified by Stephen McConnachie*

Algorithm returns 'true' if integer is even, 'false' otherwise

```
function result = iseven(x)
if mod(x,2)==0
    result=1;
    return
elseif mod(x,2)==1
    result=0;
    return
else
    error('Must have integer input')
end
```

modexp.m *code written by Hamish Silverwood*

Fast Exponentiation Algorithm, computes x^e mod m

```
function result = modexp(x, e, m)
Y=1;

while e~=0
    if iseven(e)
        x = mod((x^2), m);
        e = e/2;
    elseif ~iseven(e)
        Y = mod((x*Y), m);
        e = e-1;
    end
end

result = Y;
```

ispwr2.m *code written by Stephen McConnachie*

Algorithm returns 'true' if $y = 2^{\ell}$ for some $\ell \in \mathbb{N}$

```
function [x]=ispwr2(y)

if y<0
    error('Input must be a non-negative integer')
end

aflag=0;
while aflag==0
    if  y==2
        x=1;
        aflag=1;
        return
    elseif y==0
        x=0;
        aflag=1;
        return
    elseif y==1
        x=1;
        aflag=1;
        return
    elseif iseven(y)==1
        y=y/2;
    elseif iseven(y)==0
        x=0;
        aflag=1;
        return
    end
end
```

pollardZ.m *code written by Stephen McConnachie*

Algorithm implements Pollard's rho-Algorithm for solving DLP in $\mathbb{Z}_p$

```
function [x] = pollardZ(g,a,p,x0)

%Check number of input variables is valid
if nargin==3
    n=p-1; %the order of a primitive element in Z_p (a generator for Z_p) is p-1
    %If only 3 input variables, generate random x0
    xi=ceil(rand*n);
elseif nargin==4
    n=p-1; %the order of a primitive element in Z_p (a generator for Z_p) is p-1
    xi=x0;
else error('Incorrect number of input arguments. Calling sequence:[x] = pollardZ
(g,a,p,x0) or [x] = pollardZ (g,a,p)')
end

%Check input variables are valid
if g>n || g<0
    error('Input variable g must be an element of Z_p')
elseif a>n || a<0
    error('Input variable a must be an element of Z_p')
elseif p<1
    error('Input variable p must be greater than 0')
elseif ~isprime(p)
    error('p must be prime')
elseif xi>n || xi<1
    error('Input variable x0 must be an element of N_n')
end

%Define subsets G1, G2, G3
G1G2=round(n/3); G2G3=round(2*n/3);

y=0; aflag=0; jj=1; tol=0.1;
while aflag==0
    B=modexp(g,xi,p);
    store=[B xi y];
    ii=0;    %Set a counter value that increases by 1 with each iteration of the
             %beta algorithm
    disp('  j  B_j  x_j  y_j')
    fprintf('%3.0f %4.0f %4.0f %4.0f\n',ii,store)
    bflag=0;
    while bflag==0
        ii=ii+1; %increase counter value by 1
        %Define piecewise function (if loop) for recursive definition of x_i
        if B<=G1G2
            xi=mod(xi+1,n);
        else
            if B<=G2G3
                xi=mod(2*xi,n);
            end
        end
```

```
    %Define piecewise function (if loop) for recursive definition of y_i
    if B>G2G3
        y=mod(y+1,n);
    else
        if B>G1G2
            y=mod(2*y,n);
        elseif abs(y)<tol
            y=0;
        end
    end

    %Define B_i in terms of x_i and y_i
    B=mod(modexp(g,xi,p)*modexp(a,y,p),p);

    %Check for match with stored triplet
    if B==store(1)
        bflag=1; store=[store xi y];
        fprintf('%3.0f %4.0f %4.0f %4.0f\n',ii,B,store(4),store(5))
        break
    elseif ispwr2(ii)==1        %...Store one triplet / eight triplets
        store=[B xi y]; fprintf('%3.0f %4.0f %4.0f %4.0f\n',ii,store)
    end
end

x_i=store(2);
x_ik=store(4);
xd=x_i-x_ik;
y_i=store(3);
y_ik=store(5);
yd=y_ik-y_i;

gyn=gcd(yd,n);
if gyn==1
    x=mod(xd*invmod(yd,n),n);
    aflag=1;
else
    divtest=mod(xd,gyn);
    if divtest~=0
        error('Cannot compute index with this algorithm - gcd does not divide
         xi-xik')
    end


    if gyn<2^32
        nn=n/gyn; yy=yd/gyn; xx=xd/gyn; yyinv=invmod(yy,nn);
        z=mod(yyinv*xx,nn);
        for k=0:gyn
            x=z+k*nn;
            a_=modexp(g,x,p);
            if a_==a
                aflag=1;
                break
```

```
                end %if loop checking correct a generated by k
            end %for loop testing each k
        end %if loop preventing excessive algorithm runtime
    end %if loop calculating index

    if aflag==0
        if nargin==4
            error('Runtime error: bad x0 value. Enter different x0 value, or increase
                   tolerance in pollardZ.m line 131 "if gyn<2^32"')
        else
            jj=jj+1;
            if jj==100
                error('Too many iterations in index algorithm. Try increasing tolerance')
            else xi=ceil(rand*n); y=0;
                continue %skip back to beta algorithm
            end
        end
    end
end


%End of Script File: pollardZ.m
```

Bibliography

[1] Buchmann, Johannes A.; *Introduction to Cryptography, 2nd ed.*

[2] Garrett, Paul B.; *Making, Breaking Codes: An Introduction To Cryptology*

[3] Martin, Ben; *MATH409 Elliptic Curve Cryptography* Lecture Notes, University of Canterbury, 2006

[4] Menezes, Alfred J.; van Oorschot, Paul C.; Vanstone, Scott A.; *Handbook of Applied Cryptography*

[5] Menezes, Alfred J.; Jurisic, Aleksandar; *Elliptic curves and cryptography,* Dr. Dobb's Journal, April 1997, 23-36

[6] Menezes, Alfred J.; *Evaluation of Security Level of Cryptography: The Elliptic Curve Discrete Logarithm Problem*
http://www.ipa.go.jp/security/enc/CRYPTREC/fy15/doc/1028_ecdlp.pdf

[7] Pollard, J.M.; *Monte Carlo Methods For Index Computation,* Mathematics of Computation, Vol. 32, No. 143. (Jul., 1978), pp. 918-924.
http://links.jstor.org/sici?sici=00255718%28197807%2932%3A143%3C918%3AMCMFIC%3E2.0.CO%3B2-6

[8] Taher ElGamal; *A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*, IEEE Transactions on Information Theory, v. IT-31, n. 4, 1985, pp469–472 or CRYPTO 84, pp10–18, Springer-Verlag.

[9] Teske, E.; *Speeding Up Pollard's rho Method For Computing Discrete Logarithms,* Technical Report No. TI-1/98, Technische Hochschule Darmstadt, Darmstadt, Germany, (1998).

[10] Wiener, Michael; van Oorschot, Paul; *Parallel collision search with cryptanalytic applications*, Journal of Cryptology, **12** (1999), 1-28